Programación con Bourne Shell

Kay Tucci

June 22, 2009

1 Prefacio

La meta de este tutorial es dar una introducción rápida a las características más comunes de la programación de *scripts* en Bourne Shell. Existen otros tutoriales que muestran los aspectos más resaltantes de otros *shells* como el *csh*, *bash* o el *tcsh*. Este tutorial ¡No es una referencia completa! Se han omitido muchos detalles y es básicamente una traducción libre del tutorial Bourne Shell Programming[1] diseñado por Andrew Arensburger.

2 Justificación

Por qué hay que aprender a programar en shell, ¿cuándo se podría estar sin hacer nada o tomando un café?

Porque, en pocas palabras, es útil.

Muchas utilidades estándar (*rdist*, *make*, *cron*, etc.) le permiten especificar un comando que se ejecute en un momento determinado. Generalmente, este comando se le pasa simplemente al Bourne Shell, lo que significa que usted puede ejecutar *scripts* completos, así que usted debe elegir que hacer en ellos.

Finalmente, Unix[2], en todas sus variedades y especies, ejecuta *scripts* escritos en Bourne shell al arrancar. Si usted quiere modificar el proceso de arranque de su sistema, usted necesita aprender a leer y escribir *scripts* escritos en Bourne Shell.

3 ¿De qué se trata este curso?

Para comenzar, ¿qué es un shell? Bajo el enfoque de Unix, un shell es un intérprete de comandos, es decir, el shell lee comandos del teclado y los ejecuta.

Además, y de esto es de lo que se trata este curso, usted puede escribir un grupo de comandos en un archivo y ejecutarlos todos con una sola instrucción. Este archivo se le llama *script*

[1] href=http://ooblick.com/text/sh/

[2] Unix es una marca registrada de The Open Group.

Un Script Simple

```
#!/bin/sh
# Rota los archivos log del procmail
cd /home/fulanito/Mail
rm procmail.log.6       # Esto no es necesario
mv procmail.log.5 procmail.log.6
mv procmail.log.4 procmail.log.5
mv procmail.log.3 procmail.log.4
mv procmail.log.2 procmail.log.3
mv procmail.log.1 procmail.log.2
mv procmail.log.0 procmail.log.1
mv procmail.log procmail.log.0
```

Hay varias cosas que comentar en este ejemplo: en primer lugar la primera línea del *script* es especial y la trataremos en la próxima sección. En segundo lugar, los comentarios comienzan con el caracter (`#`) y continúan hasta el final de la línea.

Por último, el *script* en sí es solamente una serie de comandos. Este *script*, en particular, se usa para rotar archivos log y, por supuesto, que eso lo podría haber hecho a mano. Pero, yo soy flojo, y además, si lo hago a mano puedo cometer errores de escritura que realmente podrían causar muchos problemas.

4 #!/bin/sh

La primera línea de un *script* debe comenzar con los caracteres `#!`, seguidos del nombre del interprete de comandos[3], en este caso `/bin/sh`

Un *script*, como cualquier otro archivo que se puede ejecutar, necesita tener permiso de ejecución. Por lo que luego de guardar el *script* con el nombre *rotaLog* se ejecuta el comando

```
chmod +x rotaLog
```

para que el *script* sea ejecutable. Ahora se puede ejecutar el *script* mediante el comando

```
./rotaLog
```

A diferencia de otros sistemas operativos, Unix permite que cualquier programa pueda ser usado como intérprete de *scripts*. Es por esto que se escucha decir que "Hice un *script* en Bourne shell", "un *script* awk", "es un *script* en perl" o incluso un "*script* de gnuplot". Por lo tanto, es muy importante hacerle saber al Unix qué programa será el interprete del *script*.

[3] Algunas versiones de Unix permiten que exista un espacio en blanco entre los caracteres `#!` y el nombre del interprete, pero otros no lo permiten. Por lo tanto, si se quiere que el *scripts* sean portátiles, no se debe dajar este espacio en blanco.

Cuando el Unix intenta ejecutar algún archivo, lee los primeros dos caracteres[4], que en el caso de los *scripts* son `#!`, y al saber que el archivo es un *script*, el Unix continúa leyendo la línea para encontrar el programa que servirá de intérprete del *script*. Para los *scripts* escritos en Bourne Shell, el programa es `/bin/sh`. Es por esto que la primera línea de un *script* en Bourne Shell debe ser

```
#!/bin/sh
```

Luego del nombre del intérprete de comandos, se puede poner uno o varios parámetros. Algunos versiones de Unix solamente aceptan un parámetro, así que, no asuma que su *script* puede tener más de un parámetro en la primera línea.

Una vez que el Unix ha determinado qué programa actuará como intérprete del *script*, ejecutará al intérprete pasándole como argumento el nombre del *script*. De esta forma, cuando se ejecuta la instrucción

```
./rotaLog
```

equivaldría a ejecutar

```
/bin/sh ./rotaLog
```

5 Variables

El `sh` permite tener variables, de igual modo que cualquier otro lenguaje de programación. Las variables no necesitan ser declaradas. Para asignarle un valor a una variable en `sh` se usa la instrucción

VAR=*valor*

y para usar el valor de una variable ya asignada se usa

$VAR

o

${VAR}

Esta última sintaxis es útil si el identificador de la variables está seguido inmediatamente por otro texto.

[4] A estos dos caracteres, es decir, a los primeros 16 bits del archivo se les llama el *Magic Number*. Unix lo utiliza el *Magic Number* para identificar el tipo de archivo

$VAR y ${VAR}

```
#!/bin/sh
INGREDIENTE=azucar

echo Hay pan $INGREDIENTEado
echo Hay pan ${INGREDIENTE}ado
```

```
% ./avisoPan
Hay pan
Hay pan azucarado
```

Hay un único tipo de variables en `sh`: cadenas. Esto es algo limitante para el programador, pero es suficiente para casi todos los propósitos

5.1 Variables locales y de entorno

Una variable en `sh` puede ser local o de entorno. Ambas clases de variables se operan de la misma manera, la diferencia se presenta cuando un *script* ejecuta otro programa, cosa que ocurre con muchísima frecuencia.

Las variables de entorno son pasadas a todos los subprocesos, mientras que las variables locales no.

Por omisión, las variables son locales. Para convertir una variable local en variable de entorno se usa la instrucción

`export` VAR

Variables locales y de entorno

El siguiente `sh` muestra un *wrapper* para el programa *netscape.bin*

```
#!/bin/sh
NETSCAPE_HOME=/usr/local/netscape

CLASSPATH=$NETSCAPE_HOME/classes
export CLASSPATH

$NETSCAPE_HOME/bin/netscape.bin
```

Aquí, la variable `NETSCAPE_HOME` es local, mientras que la variable `CLASSPATH` es de entorno. El *script* pasa la variable `CLASSPATH` al programa `netscape.bin` (porque `netscape.bin` seguramente la usará para encontrar los archivos de las

clases de Java). En cambio, la variable `NETSCAPE_HOME` se usa solamente en el *script* y el programa `netscape.bin` no la necesita por lo que se mantiene local.

La única forma de hacer que una variable deje de ser de entorno es mediante la instrucción

```
unset VAR
```

Esta instrucción elimina la variable de la tabla de símbolos del *script*, haciendola desaparecer con el efecto colateral de que se deja de exportar[5].

Las variables que se le pasa al *script* como parte del conjunto de variables de entorno, son variables de entorno del *script* desde el inicio de su ejecución. Si se quiere evitar que el *script* pase alguna de estas variables a los programas que llama, se debe ejecutar el comando `unset` cerca del principio del *script* o antes de la llamada al programa al que no se le quiere pasar la variable.

Si un *script* hace referencia a una variable que nunca ha sido asignada el `sh` asume que su valor corresponde al de una cadena vacia.

Variables no asignadas

```
#!/bin/sh
echo Hola $NAME !
echo Chao ${NAME}!
NAME=Fulanito
echo Hola $NAME !
echo Chao ${NAME}!
```

```
% ./saludo
Hola !
Chao !
Hola Fulanito !
Chao Fulanito!
```

5.2 Variables Especiales

El `sh` trata a ciertas variables de forma especial. Algunas de estas variables especiales son asignadas automáticamente y otras afectan el comportamiento y la forma como los comandos son interpretados.

[5]También hay otros efectos colaterales, si existe una función cuyo nombre coincide con el de la variable, la instrucción `unset` borrará esa función

Variables Especiales

`$1`, `$2`...`$9`	Argumentos del *script*.
`$0`	Nombre del *script*.
`$*`, `$@`	Todos los argumentos del *script*.
`$#`	Cantidad de argumentos del *script*.
`$?`	Status de salida del último comando.
`$-`	opciones del `sh` .
`$$`	PID del proceso actual.
`$!`	PID del último proceso corriendo en *background*.
`$IFS`	Separador de campos *Input Field Separator*.

5.2.1 Los argumentos del *script*

La variable `$0` almacena el nombre del *script*. Equivale al `argv[0]` en un programa C.

De las variables especiales, las más útiles son las que referencian a los argumentos del *script*. `$1` se refiere al primer argumento luego del nombre del *script*, `$2` al segundo, y así sucesivamente hasta `$9`.

Argumentos del *script*

```
#!/bin/sh
echo El script:  $0
echo El primero: $1
echo El segundo: $2
echo El tercero: $3
```

```
% ./argumentos a b c
El script: argumentos
El primero: a
El segundo: b
El tercero: c
```

Si el *script* tiene más de 9 argumentos, se puede usar el comando `shift` para descartar el primer argumento y correr los argumentos restantes una posición.

Argumentos del *script*

```
#!/bin/sh
echo El script:  $0
echo El primero: $1; shift
echo El segundo: $1; shift
echo El tercero: $1
```

```
% ./argumentos a b c
El script: argumentos
El primero: a
El segundo: b
El tercero: c
```

En ocaciones es necesario contar con la lista completa de todos los argumentos. Para esto, `sh` proporciona dos variables `$*` y `$@` las cuales contienen todos los argumentos del *script* y equivalen a `$1 $2 ...`

La diferencia entre `$*` y `$@` se aprecia cuando están encerradas entre comillas dobles " "[6]. La variable `$*` se comporta del modo usual, por ejemplo con tres argumentos

`"$*"` equivale a `"$1 $2 $3"`

mientras que `$@` enciarra entre comillas dobles y por separado a cada uno de los argumentos, es decir que para el ejemplo anterior

`"$@"` equivale a `"$1" "$2" "$3"`

El número total de argumentos con que se llamo al *script* está almacenado en la variable `$#`.

Un buen *script* debe verificar que los argumentos sean correctos y de no serlo debe dar información sobre los mismos.

[6] El uso de las comillas se estudiará más adelante en la sección 7.3.

Verificación de Argumentos

```
#!/bin/sh
if [ $# -ne 2 ]; then
  echo 1>&2 Use: $0 nombre edad
  exit 127
fi
echo Bien ${1}!!! a los $2 ya sabes leer
exit 0
```

```
% ./verificaArgs
Use: ./verificaArgs nombre edad
% ./verificaArgs fulanito 20
Bien fulanito!!! a los 20 ya sabes leer
```

5.2.2 Las otras variables especiales

- $? Almacena el status de salida del último comando ejecutado. Cero (0) indica que la el comando se ejecutó normalmente.
- $- lista todas las opciones con las que el `sh` fue invocado. Para más detalle consulte el manual de **sh**(1)

  ```
  % man sh
  ```

- $$ conserva el PID del proceso actual.
- $! conserva el PID del último proceso que se ejecutó en *background*.
- $IFS (*Input Field Separator*) Determina cuál va a ser el delimitador para separar las cadenas en campos.

5.3 Expresiones de Quasi-variable

La expresión ${VAR} es en realidad un caso especial de un conjunto de expresiones más generales:

${VAR:-*expresión*}	Usa el valor por omisión: Si VAR está asignada y no es nula, evalúa $VAR. De lo contrario, evalúa *expresión*.
${VAR:=*expresión*}	Asigna valor por omisión: Si VAR está asignada y no es nula, evalúa $VAR. De lo contrario, asigna a VAR *expresión* y evalúa *expresión*.

`${VAR:?`*expresión*`}` Si VAR está signada y no es nula, evalúa $VAR. De lo contrario, muestra *expresión* en el salida estándar de error y finaliza con un status no nulo.

`${VAR:+`*expresión*`}` Si VAR está signada y no es nula, evalúa una cadena vacía. De lo contrario, evalúa *expresión*.

`${#VAR}` Evalúa la longitud de $VAR.

En todos los casos anteriores de determina si VAR está asignada y no es nula. Si eliminamos los dos puntos (:) solamente se verifica si VAR está asignada.

6 Patrones y Expansiones

El `sh` soporta un limitado grupo de formas de patrones y concordancias. Los operadores son:

`*` Concuerda con cero o más caracteres.

`?` Concuerda con un solo caracter.

`[`*rango*`]` Concuerda con cualquier caracter del *rango*.

Cuando una expresión contiene alguno de estos operadores se crea un patrón que el `sh` sustituye por la lista de archivos cuyos nombres se ajusten al patrón. Esto se conoce como expanción o *"globbing"* en inglés. También son muy usados en la estuctura de selección `case` (ver sección 9.4).

La regla tiene su exepción. Cuando se expande `*` o `?` los patrones no se ajustarán a los archivos que comiencen por punto (.). Para que los patrones se ajusten a ellos deben especificar el punto explicitamente. Por ejemplo: `.*` o `.??gar`

Para las personas de cierta edad: en MS-DOS, el patrón `*.*` se ajustaba a todos los archivos. En `sh`, este patrón se ajusta a todos los archivos que contienen un punto.

7 Comillas y Backslash

Si en `sh` se ejecuta

```
echo ***   MALETIN LLENO DE $$$   ***
```

El **NO** mostrará en pantalla el mensaje

```
***   MALETIN LLENO DE $$$   ***
```

Esto ocurre porque en primer lugar, el `sh` expande los asteriscos (`***`) remplazándolos por la lista de todos los archivos cuyos nombres no comienzan por punto que están el el directorio de trabajo. Luego, como las palabras se pueden

separar por cualquier cantidad de espacios en blanco y tabuladores, el `sh` comprime los tres espacios en blanco en uno solo. Finalmente, remplaza la primera ocurrencia de `$$` por el PID del `sh`. Es para estos casos que las comillas y el *backslash* entran a jugar un papel importante.

El `sh` maneja diferentes tipos de comillas, cada una con una semántica diferente. Pero comencemos por el *backslash*("\").

7.1 Backslash

Al igual que en las cadenas C, un *backslash* elimina cualquier significado especial que pueda tener el caracter que lo sigue. Si el caracter siguiente a un *backslash* no tiene ningún significado especial, el *backslash* no tiene efecto.

El mismo *backslash* es un caracter con significado especial así que para eliminár-selo se de escribir doble: \\. Así que la instrucción

```
echo \*\*\*   MALETIN LLENO DE \$\$\$   \*\*\*
```

mostrará en pantalla

```
echo *** MALETIN LLENO DE $$$ ***
```

7.2 Comillas simples

Las comillas simples funcionan como las citas textuales

```
'HOLA'
```

Cualquier cosa encerrado entre comillas simples es textual excepto una comilla simple, por supuesto.

```
echo '***   MALETIN LLENO DE $$$   ***'
```

y aparecerá en pantalla exactamento el mensaje que se quiere.

Hay que decir que el *backslash* también pierde su significado especial encerrado entre comillas simples, así que por una parte no hay que ponerlo doble para que tenga su significado normal, pero también esto hace imposible introducir una comilla simple dentro de comillas simples.

7.3 Comillas dobles

Las comillas dobles, como

```
"echo ***   MALETIN LLENO DE \$\$\$   ***"
```

preservan la mayoría de los caracteres especiales. Sin embargo, las variables y las expresiones encerradas en *backquotes* (‘), estas últimas tratadas en la sección 7.4, se remplazan por los valores correspondientes a sus expanciones.

7.4 *Backquotes* (‘)

Si en un *script* se tiene una expresión encerrada entre *backquotes* (‘), como por ejemplo,

```
‘date‘
```

la expresión será evaluada por el `sh` como un comando y se remplazará por lo que el comando escriba en la salida estándar.

***backquotes* (‘)**

```
#!/bin/sh
echo Hola ‘whoami‘, estamos a ‘date‘
exit 0
```

```
% ./fecha
Hola fulanito, estamos a Fri Nov 9 16:37:49 GMT 2007
```

8 Comandos intrínsecos

El `sh` contiene varios comandos intrínsecos o *built-in commands*, es decir, comandos que no corresponden a ningún programa. Entre los más usados tenemos:

{ ***comandos*** ; },(***comandos***) Ejecuta los *commands* en un *subshell*. Desde el punto de vista del *script* son un único comando lo que puede ser útil para redireccionar la E/S (vea la sección 10).

La forma { *comandos*; } es más eficiente porque no se crea realmente un *subshell*, lo que significa además que las variables asignandas en los *comandos* son visibles para el resto del *script*.

: **(dos puntos)** No hace nada. Generalmente se usa en

: ${VAR:=*valor por omisión*}

. ***archivo*** El comando punto “.” lee el *archivo* y si no hay problemas reemplaza la línea donde se encuentra por las instrucciones que se encuentran en el *archivo*.

`bg` [***trabajo***], `fg` [***trabajo***] `bg` ejecuta el trabajo especificado en *background*. `fg` retorna el trabajo especificado a *foreground*. En ambos casos si no se especifica el trabajo se asume que se trata del trabajo actual. Los trabajos se identifican como `%`*número*. El comando `jobs` lista los trabajos.

`cd` [***dir***] Asigna el directorio actual a *dir*. Si *dir* no se especifica, asigna el directorios hogar (*home*) como directorio actual.

`pwd` Muestra el directorio actual.

`echo` [***args***] Muestra *args* en la salida estándar.

`eval` ***args*** Evalua los argumentos como una expresión `sh`. Esto permite construir cadenas fácilmente y ejecutarlas.

`exec` ***comando*** Ejecuta el comando especificado, y remplaza el *shell* actual por ese comando. Esto significa que las instrucciones que siguen al comando `exec` no se ejecutarán a menos que el comando `exec` falle.

`exit` [***n***] Finaliza el *shell* actual con el código de salida *n*. Si *n* no se especifica, por omisión el código será cero.

`kill` [`-`***sig***] ***%job*** Envía la señal *sig* al trabajo especificado. *sig* puede ser expresada numérica o simbólicamente. `kill -l` muestra todas las señales disponibles. Por omisión la señal *sig* será SIGTERM (15).

`read` ***var...*** Lee una línea de la entrada estándar y la asigna a la variable *var*. Si se le dan varias variables var_1, var_2, var_3, etc., la primera palabra se le asigna a var_1, la segunda a var_2 y así sucesivamente hasta la peúltima variable y la parte restante se le asigna a la última variable.

`set` [`+/-`***bandera***] [***arg***] Sin argumentos muestra el valor de todas las variables.

`set -`x activa la opción x del `sh`; `set +`x desactiva la opción.

`set` *args...* asigna los argumentos de la línea de comandos con *args*.

`test` ***expresión*** Evalúa la *expresión* booleana finalizando con un código de salida cero si la es verdadera o diferente de cero si es falsa. Más detalles en la sección 12.3.

`trap` [***comando sig***]**...** Si la señal *sig* es enviada al *shell*, se ejecuta el *comando*. Es muy útil para salidas limpias, por ejemplo para remover los archivos temporales cuando el *script* es interrumpido.

`ulimit` Muestra o asigna los límites para escribir archivos.

`umask` [***nnn***] Asigna la `umask` en *nnn* (en octal). Sin argumentos muestra el valor actual de la `umask`. La mayor utilidad de este comando es para crear archivos con ciertas restricciones de lectura.

`wait` [***n***] Espera por la finalización de un proceso que está en *background* y cuyo PID *n*. Sin argumentos espera por la finalización de todos los procesos *background*.

Esta lista no es exaustiva. Además, hay que tener presente que la lista de comandos intrísecos cambia entre las diferentes implementaciones.

9 Control del flujo

El `sh` soporta las estructuras escenciales programación para el control del flujo de los *scripts*.

9.1 `if`

Al igual que en otros lenguajes de programación, la estructura `if` se usa para selección simple, doble y anidada. Su sintaxis es la siguiente

`if` *condición*$_1$ `; then`
  inst$_1$
[`elif` *condición*$_2$ `; then`
  inst$_2$
  ...]
[`else`
  inst$_n$]
`fi`

Esto es todo, un bloque `if` seguido opcionalmente por uno o más bloques `elif` (que es la abreviatura de *else if*), seguido también opcionalmente por un bloque `else`, y finalizando con un `fi`.

La estructura `if` funciona como cualquier programador espera que sea: si la *condición*$_1$ es cierta se ejecuta la instrucción o instrucciones *inst*$_1$ y se salta al `fi`. De lo contrario se evalua la siguiente condición, *condición*$_2$ y de ser cierta se ejecuta *inst*$_2$ y se salta al `fi`. De no ser cierta ninguna de las condiciones se ejecuta *inst*$_n$.Note que el constructor `elif` se utiliza simplemente para evitar que se note el anidamiento de los `if`s.

if

```
#!/bin/sh
AUTORIZADO=fulanito
NOMBRE=`whoami`

if [ $NOMBRE = $AUTORIZADO ]; then
        echo "Bienvenido al script $0"
else
        echo "Solamente $AUTORIZADO puede ejecutar $0"
        exit 1
fi
```

¿Algún problema con la sintaxis? Observen bien el ejemplo y verifique que todo esté bien. Ah!!! En la sintaxis no se pusieron los corchetes!!!

En realidad el ejemplo es sintácticamente correcto. El corchete es un comando del `sh` */bin/[* o */usr/bin/[*, y es otro nombre para el comando *test*. En la sección 12.3 se darán más detalles del comando.

La *condición* en la estructura de selección puede ser en realidad cualquier comando. Si el el comando retorna cero, la *condición* es verdadera; en cambio, si retorna cualquier cosa diferente de cero, la *condición* es falsa[7]. Por esto es posible escribir

If sin test

```
#!/bin/sh
if [ $# != 1 ]; then
  echo 1>&2 Use: login
  exit 127
fi
if grep $1 /etc/passwd; then
  echo "$1 tiene cuenta"
else
  echo "$1 no tienen cuenta"
fi
```

```
% ./ifSinTest fulanito
fulanito:x:1000:100:Fulanito:/home/fulanito:/bin/bash
fulanito tiene cuenta
```

9.2 `while`

La estructura `while` también es similar a la utilizada en otros lenguajes de programación. Su sintaxis en `sh` es:

`while` *condición*`; do`
    instucción
`done`

Como es de esperar, el lazo `while` ejecuta la o las *instruccione*s mientras la *condición* sea verdadera. Al igual que en el `if`, la *condición* puede ser cualquier comando, y su valor es verdadero o falso dependiendo de si estado de salida del comando es cero o no cero respectivamente.

Las instrucciones asociadas a un lazos `while` pueden contener los comandos `break` y `continue`.

[7] Justo al contrario de C!!!

El comando `break` interrumpe el lazo `while` inmediatamente. enviando el control del programa a la línea siguiente `done`.

El comando `continue` salta el resto de la instrucciones asociadas al `while` que se encuentran luego de él, volviendo al inicio del lazo donde la *condición* será nuevamente evaluada.

9.3 `for`

Como en `sh` las variables son solamente de tipo cadena de caracteres, la estructura de repetición `for`, es algo diferente de las usadas en otros lenguajes de programación. Ella itera sobre cada uno de los elementos de una lista. Su sintaxis es:

`for` *var* `in` *lista*`; do`
    instrucción
`done`

La *lista* puede estar vacia o contener una o más palabras. la estructura `for` en cada iteración le asigna a la variable *var* uno de las palabras, siguiendo el orden de aparición de éstas en la *lista* para luego ejecutar la o las *instruccio*es asociadas a él.

Por ejemplo:

for

```
#!/bin/sh
for d in Escritorio tmp "Mis Documentos"; do
  echo "\$d: $d"
done
```

```
% ./forTest
$d: Escritorio
$d: tmp
$d: Mis Documentos
```

Las instrucciones asociadas al lazo `for` también pueden tener los comandos `break` y `continue`. Ambos comando actúan igual que en el caso del `while`.

9.4 `case`

La estructura `case` funciona de la misma forma que el `switch` de C, excepto por que la concordancia se realiza con patrones y no con valores numéricos. Su sintaxis es:

```
case expresión in
     patrón)
             intrucción
             ;;
     ...
esac
```

donde, el valor de *expresión* es el de una cadena de caracteres, que por lo general viene de una variable o de la ejecución de un comando usando las *backquote*s.

El *patrón* puede usar los caracteres especiales que se vieron en la sección 6.

La concordancia con los patrones se realiza en el orden en que estos últimos aparecen, y solamente las instrucciones asociadas al primer patrón que concuerde serán ejecutadas[8]. Frecuentemente, se quiere incluir instrucciones que se ejecutarán si ningún patrón concuerda, para hacer esto se incluye el patron `*` como último patrón y se le asocian a este patrón estas instrucciones.

10 Redireccionamiento de la E/S

Las entradas y salidas de los comandos pueden ser redireccionadas a otros comandos o a archivos. Por omisión, cada proceso tienen tres tres descriptores de archivos para el manejo de E/S: (0) es el de la entrada estándar, (1) el de la salida estándar y (2) el de la salida estándar de los errores. A menos que se redireccionen, el primero corresponde al teclado y los dos últimos a la pantalla del terminal.

Sin embargo, se puede hacer el redirecionamiento de uno o más de los descriptores:

`<` ***archivo*** Conecta la entrada estándar al *archivo*. Esto permita que el comando obtenga su entrada del *archivo*, y no desde el teclado.

`>` ***archivo*** Conecta la salida estándar al *archivo*. Esto permite almacenar la salida de un comando en el *archivo*. Si el *archivo* no existe, es creado, y si existe, es vaciado antes de almacenar la salida en él.

`>>` ***archivo*** Conecta la salida estándar al *archivo*. Pero, a diferencia de `>`, si el *archivo* existe, la salida del comando se agrega al final de él.

`<<`***palabla*** Esta estructura actualmente no es tan usada como pudiera serlo. Provoca que la entrada estándar del comando se lea del mismo `sh` hasta que encuentre la *palabra*. No debe haber espacios entre `<<` y la *palabra*.

Por ejemplo, de esta forma el *script* puede escribir un programa:

[8] En esto se diferencia de C, donde hace falta un `break` para que las instrucciones de los otros patrones no se ejecuten.

```
#!/bin/sh
cat > holaMundo.c <<EOT
#include <stdio.h>

main()
{
  printf("Hola mundo!\n");
}
EOT
```

También es útil para escribir mensajes de varias líneas, por ejemplo:

```
linea=13
cat <<EOT
Un error ha ocurrido en la linea $linea.
Para mas informacion lea el manual de programa.
EOT
```

Como se ve es este ejemplo, que el `<<` actúa como una comillas ‘‘‘‘acts like double, es decir que las variables son expandidas. Sin embargo, si la *palabra* se encierra entre comillas simples o dobles, el `<<` actuar como comillas simples.

`<&`*dígito* Usa el descriptor de archivo *dígito* como entrada estándar.

`>&`*dígito* Usa el descriptor de archivo *dígito* como salida estándar.

`<&-` Cierra la entrada estándar.

`>&-` Cierra la salida estándar.

***comando*$_1$ `|` *comando*$_2$** Crea un conducto o *pipe* desde el *comando*$_1$ hacia el *comando*$_2$: la salida estándar del *comando*$_1$ se conecta a la entrada estándar de *comando*$_2$. Funcionalmente actúa de la misma forma que:

comando$_1$ `> /tmp/temporal`
comando$_2$ `< /tmp/temporal`

con la salvedad de que el archivo `/tmp/temporal` no se crea y que ambos comando pueden ejecutarse simultaneamente[9].

Cualquier número de comandos puede conectarse con *pipes*.

***comando*$_1$ `&&` *comando*$_2$** Ejecuta el *comando*$_1$ y si su estado de salida es cero, es decir, es verdadero, ejecuta el *comando*$_2$.

***comando*$_1$ `||` *comando*$_2$** Ejecuta el *comando*$_1$ y si su estado de salida no es cero, es decir, es falso, ejecuta el *comando*$_2$.

[9] Hay un dicho que dice, *“Un archivo temporal es solamente una conexión (pipe) con actitud y deseos de vivir.”*

Si una estructura de redirecionamiento de E/S es precedida por un dígito, la estructura se aplica al descriptor de archivo correspondiente al dígito y no al descriptor por omisión. Por ejemplo:

comando `2>&1 >` *archivo*

con `2>&1` asocia el descriptor de archivo 2, es decir la salida estándar de errores, con el descriptor de archivo 1, que corresponde a la salida standar, y luego el `>` *archivo* redireciona la salida estándar al *archivo*.

Esta estructura es útil para escribir mensajes de errores:

```
echo "A $usuario se le nego el premiso" 1>&2
```

Las estructuras de redirecionamiento se precesan de izquierda a derecha.

11 Funciones

Cuando un conjunto de instrucciones se repiten varias veces en un *script* puede ser útil definir una función que las agrupe. Definir una función se asemeja a crear un *subscript* dentro del *script*. Una función e define así:

nombre `()` {
    comandos
}

y se llama como cualquier otro comando:

nombre args...

Con las funciones se puede, por ejemplo, redireccionar la E/S, encerrarla entre *backquotes* o hacer cualquier cosa que se le haga a un comando.

A pesar de esto, las funciones no son *scripts* y por lo tanto al llamarlas no se despliega un *subshell* para ejecutarlas. esto significa que si una variable se asigna dentro de la función, la variable tendrá ese valor luego que la función finalice y el control de la ejecución vuelva al programa que llamo a la función.

Las funciones pueden usar la instrucción `return` *n* para finalizar con un estado de salida igual a *n*. También la función puede terminar con un `exit` *n*, pero esto provoca que tanto la función como el *script* finalicen su ejecución.

11.1 Argumentos de las functiones

Las funciones toman los argumentos de la misma forma que lo hacen los *scripts* con los argumentos de la línea de comando. Los argumentos están almacenados en las variables especiales $1, $2, ... $9; al igual que ocurre en los *scripts*.

12 Utilidades externas

Existe un gran número de comandos que no forman parte del `sh`, pero son utilizadas frecuentemente en los *scripts*. Estas son algunos de ellos:

12.1 basename

`basename` *camino*

Escribe la última parte del *camino*. Por ejemplo:

```
basename /usr/local/bin/gnuplot
```

escribe

```
gnuplot
```

12.2 dirname

`dirname` *camino*

Es el complemento `basename`. Escribe todo el *camino* menos la última parte, es decir, si el *camino* referencia a un archivo, `basename` muestra el directorio que lo contiene. Por ejemplo:

```
dirname /usr/local/bin/gnuplot
```

prints

```
/usr/local/bin
```

12.3 [

El `/bin/[` es otro nombre para el `/bin/test`. Este comando evalúa sus argumentos como una expresión booleana, y de ser cierta el código de salida será cero (0) y de ser falsa el código será uno (1).

Si el `test` se invoca con su forma `[`, entonces el último argumento debe ser un corchete que cierra, `]`.

El `test` comprende, entre otras, las siguientes expresiones: others:

`-e` ***archivo*** Verdadero si el *archivo* existe.

`-d` ***archivo*** Verdadero si el *archivo* existe y es un directorio.

`-f` ***archivo*** Verdadero si el *archivo* existe y es un archivo plano.

`-h` ***archivo*** Verdadero si el *archivo* existe y es un enlace simbólico.

`-r` ***archivo*** Verdadero si el *archivo* existe y es legible.

`-w` ***archivo*** Verdadero si el *archivo* existe y permite escrituras.

`-n` ***cadena*** Verdadero si el tamaño de la *cadena* es no nula.

`-z` ***cadena*** Verdadero si la longitud de *cadena* es cero.

cadena Verdadero si la *cadena* no es una cadena vacía.

s1 `=` ***s2*** Verdadero si las cadenas *s1* y *s2* son idénticas.

s1 `!=` ***s2*** Verdadero si las cadenas *s1* y *s2* no son idénticas.

n1 `-eq` ***n2*** Verdadero si los números *n1* y *n2* son iguales.

n1 `-ne` ***n2*** Verdadero si los números *n1* y *n2* no son iguales.

n1 `-gt` ***n2*** Verdadero si el número *n1* es mayor que *n2*.

n1 `-ge` ***n2*** Verdadero si el número *n1* es mayor o igual que *n2*.

n1 `-lt` ***n2*** Verdadero si el número *n1* es menor que *n2*.

n1 `-le` ***n2*** Verdadero si el número *n1* es menor o igual que *n2*.

`!` ***expresión*** Niega el valor de la *expresión*, esto es, devielve verdadero si *expresión* es falsa y falso si *expresión* es verdadera.

expr1 `-a` ***expr2*** Verdadero si ambas expresiones, *expr1* y *expr2* son verdaderas.

expr1 `-o` ***expr2*** Verdadero si cualquiera de las expresiones, *expr1* o *expr2* son verdaderas.

`(` ***expresión*** `)` Verdadero si la *expression* es verdadera. Esto permite agrupar y anidar expresiones.

El comando `test` no permite evaluaciones en cortocircuito, porque todos sus argumentos son evaluados por el `sh` antes de pasar al `test`. Si se quiere tener los beneficios de la evaluación en cortocircuito, el *script* debe anidar las estructuras `if`.

12.4 echo

El comando `echo` es intrínseco en la mayoría de las implementaciones de `sh`, pero generalmente también existe el comando externo.

El `echo` solamente escribe sus argumentos en la salida estándar. Puede usarse las opciones para que no agrege al final de sus argumentos el caracter de línea nueva. En los Unix tipo BSD se escribe:

```
echo -n "cadena"
```

y en los Unix tipo System-V:

```
echo "cadena\c"
```

12.5 awk

El *Awk* y sus derivados, *nawk* y *gawk*; son lenguajes completos para escribir *scripts*. Dentro de los *scripts* en `sh` son usados generalmente por su habilidad en picar las líneas de entrada en campos, operar con esos campos y escribir los resultados. por ejemplo, el siguiente comando `awk` lee el archivo */etc/passwd* e imprime en la salida el login y el `uid` de cada usuario:

```
awk -F : '{print $1, $3 }' /etc/passwd
```

La opción `-F :` indica que el separador de campos en las líneas de entrada es el caracter dos puntos `:`. Por omisión, *awk* utiliza los espacios en blanco como separador de campos.

12.6 sed

El comando *Sed* (*stream editor*) es también un lenguaje completo para escribir *scripts*, aunque de menor alcance y más complicado que el *awk*. En los `sh` *scripts*, el *sed* es usado principalmente para realizar sustituciones. El siguiente comando lee la entrada estándar y sustituye cada ocurrencia de "Hello" por "Hola", y escribe el resultado en la salida estándar:

```
sed -e 's/Hello/Hola/g'
```

El último campo `g` indica que se realizen la sustitución en todas las ocurrencias de "Hello" por "Hola"en cada línea. Sin esta opción, solamente se cambia la primera ocurrencia de "Hello" por "Hola".

12.7 tee

El comando `tee` [`-a`] *archivo* lee la entrada estándar y la copia en la salida estándar y simultaneamente la almacena en el *archivo*.

Por omisión, `tee` vacía el *archivo* antes de comenzar a almacenar. Con la opción `-a` la salida se le agrega al *archivo*.

12.8 otros comandos

`cat`

`cut`

`grep`

`ls`

`tr`

`file`

`find`

`nice`

`head`

`tail`

13 Debugging

Desafortunadamente para los `sh` *scripts* no existen depuradores simbólicos como el *gdb*. Unfortunately, there are no symbolic debuggers such as *gdb* for `sh` scripts. When you're debugging a script, you'll have to rely the tried and true method of inserting trace statements, and using some useful options to `sh`:

La opción `-n` hace que el `sh` lea el *script* pero no lo ejecute. Esta opción es útil para verificar la sintaxis del *script*.

La opción `-x` hace que el `sh` muestre cada comando que se ejecuta en la salida estándar de los errores. Como usar esta opción al llamar al *script* puede generar mucha información, es conveniente activarla justo antes de la sección del *script* donde se quiere depurar el código y luego desactivarla:

```
set -x
# Codigo con problemas
grep $usuario /etc/passwd 1>&2 > /dev/null
set +x
```

14 Estilo

A continuación se daran algunos *tips* de estilo, asi como también algunos trucos que pueden ser útiles.

14.1 Escriba *scripts* simples y lineales

La principal ventaja del `sh` es su portatibilidad, se encuentra en todas las especies de Unix y las implementaciones son "razonablemente" estándar. Además, se pueden escribir *scripts* en `sh` que realicen casi todas las actividades. Sin embargo, los *scripts* en `sh` son bastante lentos y no existen buenas herramientas para su depuración.

Por esto último es mejor mantener los códigos tan simples y lineales como sea posible, es decir, que si un *script* tiene muchos lazos y estructuras de decisión anidadas, o necesita complicados *scripts* en *awk*; probablemente lo mejor sea reescribir el programa en *Perl* o C.

14.2 Variables de configuración al inicio

Si es posible de preveer modificaciones o cambios en la configuración del *script*, es conveniente que las variables que lo permiten estén definidas al inicio del código. Por ejemplo, si el *script* necesita ejecutar la máquina virtual de Java, *java*, se puede escribir

```
#!/bin/sh
... y 300 líneas después ...
/usr/local/jdk-1.6.0_r6/bin/java Main
```

Sin embargo, la versión de *jdk* puede cambiar u otra persona lo puede tener instalado en un sitio diferente, así que es mejor escrobir

```
#!/bin/sh
JAVA=/usr/local/jdk-1.6.0_r6/bin/java
...y 300 líneas después ...
$JAVA Main
```

14.3 No abuse de las funciones

Las funciones son fabulosas y dan orden y claridad, pero `sh` no es C ni Pascal ni Fortran. En particular, no se conveniente encapsular todo con funciones y hay que evitar que funciones llamen a otras funciones porque no es fácil ni agradable depurar un *script* que tiene llamadas a funciones de nivel 5, 6 o más.

14.4 Cadenas multilíneas

Recuerde que se pueden escribir caracteres de fin de línea en cadenas encerradas por comillas simples o dobles. Por esto escriba los mensajes con múltiples líneas usando esta facilidad.

14.5 Use : ${VAR:=valor} y asigne valores por omisión

Supongamos que se tiene un *script* que le permite al usuario editar un archivo. Esto se puede programar incluyendo la línea

```
vi $archivo
```

en el *script*. Pero, supongamos también que el usuario prefiere utilizar un editor *Emacs*, entonces, como en su sistema existe una variable de entorno `$EDITOR` que cada usuario configura se puede sustituir la línea anterior por

```
$EDITOR $archivo
```

Sin embargo esto puede causar problemas si la variable `$EDITOR` no ha sido asignada. Así que, es mejor escribir

```
: ${EDITOR:=vi}
$EDITOR $archivo
```

asignandole a `$EDITOR` un valor por omisión razonable en caso de que no haya sido asignado previamente.

15 Paranoia

Al igual que en cualquier otro lenguaje de programación, es muy fácil escribir *scripts* en `sh` que no hagan los que uno quieren sino que uno dijo que hicieran, así que, una dosis razonable de paranoia no está mal. En particular, los *scripts*

que usen datos proporcionados por el usuario deben ser capaces de manipular cualquier tipo de entrada. Por ejemplo, es casi seguro que a un *script* cgi-bin no sólo le le serán dados datos de entrada incorrectos, sino que también habra entradas maliciosas. Por otro lado, errores en *script* que se ejecuten con privilegios de root o bin pueden causar daños inimaginables al sistema y a los usuarios del mismo.

15.1 Uso del setuid en los *scripts*

NO LO HAGA!!!

Como se vió en la sección 4, los *scripts* se ejecutan de la siguiente forma: Unix abre el archivo donde está escrito el *script* y leyendo la primera línea determina que es un *script* y cuál programa es su intérprete. Entonces Unix invoca al intérprete y le pasa el nombre del *script* como argumento en la línea de comandos. Por último el intérprete abre el archivo, lo lee y lo ejecuta.

De lo anterior, se puede ver que hay un retraso entre cuando el OS abre *script*, y cuando el intérprete lo abre. Esto significa que hay una condición que un atacante puede explotar: creando un enlace simbólico que apunte al *script* con setuid; y entonces, después de que el OS haya determinado el interpeter, pero antes de que el intérprete abre el archivo, substituya ese enlace simbólico por otro *script* de su preferencia. ¡Y Listo! ¡Tenemos un `sh` con privilegios de root!

Este problema es inherente a la forma como opera el SO, y por lo tanto no es fácil resolver este problema.

Los programas compilados no presentan este problema, así el archivo *a.out*, un ejecutable compilado, al ejecutarse no se cierra para luego reabrirse, sino que el SO lo carga en memoria. Por esto, si la aplicación necesita hacer un *setuid* y es más facíl programarla con un *script* en `sh` una solución es hacerle un *wrapper* en C donde simplemente se ejecuta la instrucción *exec* para ejecutar el *script*. En este caso, hay que estar atento de todos los problemas que tiene el hacer un *setiud* y por lo tanto hay que ser paranoico al escribir el programa, pero los problemas que se originan por la doble lectura del *script* están resueltos.

15.2 `$IFS`

Uno de las primeras instruccines de un *script* debe ser

```
IFS=
```

que se encarga de reasignar el separador de campos de entrada, *Input Field Separator* a su valor por omisión. De no hacerse esto, el *script* puede heredar el `$IFS` definido por el usuario, quien pudo asignarle cualquier valor extraño para que el `sh` se comportara de forma diferente a la habitual, y esto puede producir que el *script* presente comportamientos muy extraños.

15.3 $PATH

Al igual que se asigna la `$IFS`, hay que asegurarse, mediante la asignación de la variable `PATH`, que el *script* encuentre los ejecutable y estos sean los correctos.

Particularmente, el usuario puede estar usando en su `PATH` el punto "." como primer elemento, y si tiene programas con nombres como *ls* o *grep* en el directorio desde donde llama al *script* el resultado será un desastre.

Como norma general nunca ponga al "." ni otra referencia relativa en la variable `PATH`.

Hay programadores que prefieren poner

```
PATH=
```

al inicio de sus *scripts* y luego agregar los directorios que sean necesarios. Si sólo los necesarios.

15.4 Variables y comillas

Es importante recordar que los valores almacenados en las variables pueden contener, por accidente o porque realmente lo necesitan, espacios en blanco u otros caracteres especiales. Para evitar problemas con estos caracteres, en el *script* hay qe asegurarse que las variables que sean interpretada como una única palabra o aquellas que contengan caracteres especiales estén encerrada entre comillas dobles. Por ejemplo, cualquier entrada y cualquier variable derivada de una entrada.

Sin las comillas dobles se puede presentar este problema

Variables sin comillas

```
#!/bin/sh
PATH=/usr/bin
DIR=Mis Documentos
file ${DIR}/datos
```

```
% ./sinComillas
Mis:              cannot open ‘Mis’ (No such file or dire ...
Documentos/Datos: cannot open ‘Documentos/Datos’ (No such ...
```

Pero al ponerle las comillas

Variables con comillas

```
#!/bin/sh
PATH=/usr/bin
DIR="Mis Documentos"
file "${DIR}"/datos
```

```
% ./conComillas
Mis Documentos/datos: ASCII text
```

Una buena práctica puede ser ponerle comillas dobles a todas las variables a menos que uno esté seguro de que no tendran espacios ni otros caracteres especiales.

15.5 Variables posiblemente no asignadas

Hay que tener presente que las variables pueden no estar asignadas o tener asignada una cadena vacía. Por ejemplo, en el código

```
if [ $respuesta = si ]; then
```

puede que `$respuesta` este asignada a una cadena nula, así que el `sh` ejecutará `if [ = si ]; then`, lo que causará un error. Es mejor escribir

```
if [ "$respuesta" = si ]; then
```

Pero el peligro en este caso es que la variable `$respuesta` sea `-f`, en cuyo caso el `sh` ejecutará `if [ -f = si ]; then`, lo que causará otro error.

Para evitar ambos errores es mejor escribir

```
if [ x"$respuesta" = xsi ]; then
```

y el problema estará resuelto.

16 Y el C shell?

El C shell, *csh*, y su variante *tcsh* es un agradable interprete interactivo, pero muchos programadores lo consideran deficiente para escribir sus *scripts*. Para mayores detalles consulte el artículo de Tom Christiansen, "Csh Programming Considered Harmful".